Compilers summer 2009
Project Part 1
Alan Ritacco

Contents

1. Introduction

This project is the creation and testing towards the formation of a complier for the programming language called Javalet, which is a subset of the Java [(1)] programming language. In project phase I we have created a scanner written using the fast lexical analyzer (Flex [(2)]) parlance, which is based off of Lex [(3)]. We are using Flex to read the symbols, based off of the tokens listed in section 2.1, to lexicographically scan our Flex sources shown in 2.2. The information for part one of this project can be found by browsing to the following URL:

http://web.cs.wpi.edu/~kal/courses/compilers/project/projectlexernew1.doc, the complete project description can be found by browsing to the following URL http://web.cs.wpi.edu/~kal/courses/compilers/project/.

2. The Scanner

The scanner portion of this project has been implemented to "scan" for valid tokens from the Javalet language. The tokens for the Javalet language, listed below, have been created in a scanner file and implemented using Flex's constructs. When running Flex against this scanner file the Flex program output will be a C program which can then be compiled to a binary. We use this binary to scan against our Javalet language file(s) and then be used as our lexical scanner for Javalet. The tokens for the Javalet language are listed below in section 2.1. We have used Flex version 2.5.35, and GCC [4] version 4.3 for the lexicographical and compilers respectively. A GNU [5] Makefile [6] has been created, for convenience, to help with the creation of the Flex output files and the resulting scanner binary file.

2.1 The Tokens

The following are the tokens as given for phase I of the project as taken from the class assignment for project phase I.

Type

"void", "int", "real"

Logical Operators

"!", "||", "&&", "!=", "==", "<", ">", "<=", ">="

Numerical Operators

"+", "-", "*", "/", "="

Punctuation

"{", "}", "(", ")", ",", ";"

Keywords

"if", "else", "while", "do", "for"

Names

Letter (Letter | Digit |_) * where a Letter is either an uppercase or lowercase letter and Digit is one of the digits from 0-9. There cannot be 2 consecutive underscores.

Integers

Sequences of 1 or more digits

Reals

Plus or minus followed by a number of digits followed by a dot ".", followed by a number of digits. Either the sequence before the dot can be null or the sequence after the dot can be null, but not both.

2.2 The Lex File

```
%{
#include <stdio.h>
%}
DIGIT    [0-9]
LETTER   [a-zA-Z]

%%
void            printf("Identifier, %s\n",yytext);
int             printf("Identifier, %s\n",yytext);
real            printf("Identifier, %s\n",yytext);
\!              printf("Logical_Operator, NOT\n");
\|\|            printf("Logical_Operator, OR\n");
\&\&            printf("Logical_Operator, AND\n");
==              printf("Logical_Operator, EQUAL\n");
\!=             printf("Logical_Operator, NOT_EQUAL\n");
\<              printf("Logical_Operator, LESS_THAN\n");
\<=             printf("Logical_Operator, LESS_OR_EQUAL\n");
>               printf("Logical_Operator, GREATER_THAN\n");
>=              printf("Logical_Operator, GREATER_OR_EQUAL\n");
\+              printf("Numerical_Operator, PLUS\n");
\-              printf("Numerical_Operator, MINUS\n");
\*              printf("Numerical_Operator, MULTIPLY\n");
\/              printf("Numerical_Operator, DIVIDE\n");
\=              printf("Numerical_Operator, EQUAL\n");
\{              printf("Punctuation, OPEN_BRACE\n");
\}              printf("Punctuation, CLOSE_BRACE\n");
\(              printf("Punctuation, OPEN_PARAN\n");
\)              printf("Punctuation, CLOSE_PARAN\n");
\,              printf("Punctuation, COMMA\n");
;               printf("Punctuation, SEMICOLON\n");
if              printf("Keyword, IF\n");
else            printf("Keyword, ELSE\n");
while           printf("Keyword, WHILE\n");
do              printf("Keyword, DO\n");
for             printf("Keyword, FOR\n");

{LETTER}(\_?(({LETTER}|{DIGIT})+\_?))*        printf("Names, %s\n",yytext);
{DIGIT}+        printf("Integer, %s\n",yytext);
[+-]{DIGIT}+.{DIGIT}+   printf("Real, %s\n",yytext);
[ \t]+          /* ignore whitespace */;
\n|\r|\r\n|\n\r    /* ignore carriage returns, and line feeds */;

%%
int main (void) {yylex(); return 0;}
int yywrap (void) {return 1;}
```

2.3 Inputs

The testing files used are number input 1-4. The Input files 1-3 are those as taken from the test cases in project phase I. Test case input 4 includes test for the following (not included in tests 1-3): double underscore input, upper case input, OR operator input, do while input, the c ++ operator input, quote's " ", escape sequence \n, and a positive real in the input. Input not recognized by the scanner at this point is ignored and outputted.

---------- Input 1 ----------

```
void input_a() {
  integer a, bb, xyz, b3, c, p, q;
  real b;
  a = b3;
  b = -2.5;
  xyz = 2 + a + bb + c - p / q;
  a = xyz * ( p + q );
  p = a - xyz - p;
}
```

------ Output for Input1 -------

```
Identifier, void
Names, input_a
Punctuation, OPEN_PARAN
Punctuation, CLOSE_PARAN
Punctuation, OPEN_BRACE
Names, integer
Names, a
Punctuation, COMMA
Names, bb
Punctuation, COMMA
Names, xyz
Punctuation, COMMA
Names, b3
Punctuation, COMMA
Names, c
Punctuation, COMMA
Names, p
Punctuation, COMMA
Names, q
Punctuation, SEMICOLON
Identifier, real
Names, b
Punctuation, SEMICOLON
Names, a
Numerical_Operator, EQUAL
Names, b3
Punctuation, SEMICOLON
Names, b
```

Numerical_Operator, EQUAL
Real, -2.5
Punctuation, SEMICOLON
Names, xyz
Numerical_Operator, EQUAL
Integer, 2
Numerical_Operator, PLUS
Names, a
Numerical_Operator, PLUS
Names, bb
Numerical_Operator, PLUS
Names, c
Numerical_Operator, MINUS
Names, p
Numerical_Operator, DIVIDE
Names, q
Punctuation, SEMICOLON
Names, a
Numerical_Operator, EQUAL
Names, xyz
Numerical_Operator, MULTIPLY
Punctuation, OPEN_PARAN
Names, p
Numerical_Operator, PLUS
Names, q
Punctuation, CLOSE_PARAN
Punctuation, SEMICOLON
Names, p
Numerical_Operator, EQUAL
Names, a
Numerical_Operator, MINUS
Names, xyz
Numerical_Operator, MINUS
Names, p
Punctuation, SEMICOLON
Punctuation, CLOSE_BRACE

----------Input 2 ----------

```
void input_b() {
  if ( i > j )
    i = i + j;
  else if ( i < j )
    i = 1;
}
```

------ Output for input 2 -----

```
Identifier, void
Names, input_b
Punctuation, OPEN_PARAN
Punctuation, CLOSE_PARAN
Punctuation, OPEN_BRACE
Keyword, IF
Punctuation, OPEN_PARAN
Names, i
Logical_Operator, GREATER_THAN
Names, j
Punctuation, CLOSE_PARAN
Names, i
Numerical_Operator, EQUAL
Names, i
Numerical_Operator, PLUS
Names, j
Punctuation, SEMICOLON
Keyword, ELSE
Keyword, IF
Punctuation, OPEN_PARAN
Names, i
Logical_Operator, LESS_THAN
Names, j
Punctuation, CLOSE_PARAN
Names, i
Numerical_Operator, EQUAL
Integer, 1
Punctuation, SEMICOLON
Punctuation, CLOSE_BRACE
```

----------Input 3----------

```
void input_c() {
  while ( i < j && j < k ) {
    k = k + 1;
    while ( i == j )
      i = i + 2;
  }
}
```

----- Output for Input 3 ----

Identifier, void
Names, input_c
Punctuation, OPEN_PARAN
Punctuation, CLOSE_PARAN
Punctuation, OPEN_BRACE
Keyword, WHILE
Punctuation, OPEN_PARAN
Names, i
Logical_Operator, LESS_THAN
Names, j
Logical_Operator, AND
Names, j
Logical_Operator, LESS_THAN
Names, k
Punctuation, CLOSE_PARAN
Punctuation, OPEN_BRACE
Names, k
Numerical_Operator, EQUAL
Names, k
Numerical_Operator, PLUS
Integer, 1
Punctuation, SEMICOLON
Keyword, WHILE
Punctuation, OPEN_PARAN
Names, i
Logical_Operator, EQUAL
Names, j
Punctuation, CLOSE_PARAN
Names, i
Numerical_Operator, EQUAL
Names, i
Numerical_Operator, PLUS
Integer, 2
Punctuation, SEMICOLON
Punctuation, CLOSE_BRACE
Punctuation, CLOSE_BRACE

---------- Input 4: An Example of your own ----------

```
int test(int x,y)
{
        int X,Y__Z;
       X=+4.5;
       Y__Z=-1.2;

        if (x || z)
                x=y;
        else
                x=2;
        x=5, z=5;
        do while(x<6)
        {
                x++;
        }
}
```

---- Output for input 4 ----

```
Identifier, int
Names, test
Punctuation, OPEN_PARAN
Identifier, int
Names, x
Punctuation, COMMA
Names, y
Punctuation, CLOSE_PARAN
Punctuation, OPEN_BRACE
Identifier, int
Names, X
Punctuation, COMMA
Names, Y
__Names, Z
Punctuation, SEMICOLON
Names, X
Numerical_Operator, EQUAL
Real, +4.5
Punctuation, SEMICOLON
Names, Y
__Names, Z
Numerical_Operator, EQUAL
Real, -1.2
Punctuation, SEMICOLON
Keyword, IF
Punctuation, OPEN_PARAN
Names, x
```

Logical_Operator, OR
Names, z
Punctuation, CLOSE_PARAN
Names, x
Numerical_Operator, EQUAL
Names, y
Punctuation, SEMICOLON
Keyword, ELSE
Names, x
Numerical_Operator, EQUAL
Integer, 2
Punctuation, SEMICOLON
Names, x
Numerical_Operator, EQUAL
Integer, 5
Punctuation, COMMA
Names, z
Numerical_Operator, EQUAL
Integer, 5
Punctuation, SEMICOLON
Keyword, DO
Keyword, WHILE
Punctuation, OPEN_PARAN
Names, x
Logical_Operator, LESS_THAN
Integer, 6
Punctuation, CLOSE_PARAN
Punctuation, OPEN_BRACE
Names, x
Numerical_Operator, PLUS
Numerical_Operator, PLUS
Punctuation, SEMICOLON
Names, printf
Punctuation, OPEN_PARAN
"Names, This
Names, is
Names, a
Names, test
\Names, n
"Punctuation, CLOSE_PARAN
Punctuation, SEMICOLON
Punctuation, CLOSE_BRACE
Punctuation, CLOSE_BRACE

2.4 Outputs

<Your outputs. Be sure to state which input the output is for. You can combine 2.3 and 2.4 if you wish and put the output right after the input.>

Input and Output for the tests are shown in 2.3 above.

Bibliography

1. **Microsystems, Sun.** Java. *The Java History Timeline.* [Online] Sun Microsystems, 1991. http://www.java.com/en/javahistory/timeline.jsp.

2. **t, Vern Paxson.** flex: The Fast Lexical Analyzer. *FLEX.* [Online] GNU, 1987.

3. **Schmidt, M. E. Lesk and E.** LEX – Lexical Analyzer Generator. *LEX.* [Online] http://opengroup.org/onlinepubs/007908775/xcu/lex.html.

4. **Stallman, Richard M.** GNU GCC. *GNU GCC.* [Online] 1987. http://gcc.gnu.org/wiki/History.

5. **Stallman, Richard.** The GNU Project. [Online] 1984. http://www.gnu.org/gnu/thegnuproject.html.

6. **License, GNU Free Documentation.** GNU Make. *GNU.* [Online] 1988. http://www.gnu.org/software/make/manual/make.html#Top.